MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers

Holger Lange, Thorsten Wink, Andreas Koch
Embedded Systems and Applications Group
Technische Universität Darmstadt, Germany
E-Mail: {lange, wink, koch}@esa.cs.tu-darmstadt.de

Abstract—**We describe a parameterized memory system suitable as target for automatic high-level language to hardware compilers for reconfigurable computers. It fully supports the spatial computation paradigm by allowing the realization of each memory operator by a dedicated hardware memory port. Inter-port coherency is maintained only for those ports that actually require it, and efficient speculative execution is enabled by a dynamic scheme for arbitrating access to shared resources (such as main memory), relying on techniques inspired by the branch prediction of conventional software-programmable processors.**

I. Introduction

Adaptive computing systems (ACS) combine the potential performance of a reconfigurable computing unit (RCU), realizing hardware accelerators (HA), with the flexibility of a conventional general-purpose processor (GPP) to handle administrative tasks (e.g., I/O, operating system, etc.) [14].

In order to simplify the ACS programming, high-level-language to ACS compilers have received much attention. In contrast to a human designer, these automatic flows always generate a system according to a predefined architecture template. This template often encompasses the usual model of separate control units (statically or dynamically scheduled) and data paths, but also has to include an interface between RCU and GPP, and, for systems aiming for high performance, directly between RCU and memory. By exploiting the reconfigurability of the platform, however, we can now also adapt the characteristics of the memory system to the specific needs of each ACS application.

Thus, the memory system itself should be highly parameterizable. Customizable characteristics include, e.g., the nature of accesses (random or sequential), architecture parameters (cache organization, streaming buffer size, prefetch policy), but also the number of parallel memory ports. The latter is especially crucial to fully support the paradigm of spatially distributed computation, which distinguishes algorithm execution in hardware on the RCU from that of the temporally distributed one of software on the GPP.

Building on our experience with the Memory Architecture for Reconfigurable Computers (MARC I) [15], in this work we present the significantly extended and refined solution MARC II.

II. System Overview

Fig. 1 gives an overview of the architecture and system integration. MARC II sits between the data path and the actual SoC infrastructure including the DRAM memory controller (n) and the processor bus (m). The details of this latter interface are encapsulated in a so-called *technology module* (TechMod, l), which isolates the higher abstraction levels of the memory system from the low-level hardware details. MARC II can be moved to a new platform by just providing an appropriate TechMod. The data path (a, which we anticipate to be generated by automatic compilers, but which can as well be manually implemented) accesses MARC II through so-called *front-ends*. We currently provide two kinds of front-ends, which differ in the semantics of data access: Cache Ports support random accesses and will be discussed in greater detail in Sec. IV, Stream Ports (not discussed here, but see [7]) have DMA-engine like programmability to implement various kinds of streaming accesses (e.g., non-unit stride, prefetched, etc.).

Internally, all Cache Ports are partitioned into Coherence Clusters (CC). Each CC may have at most one Write (d) and one or more Read Ports (c). All Ports in a cluster can potentially affect memory in overlapping address ranges and are kept coherent using a cluster-wide Coherency Bus (CB, f) and its associated protocols. Each Port is backed by its own *dedicated* cache memory (e), resulting in improved scalability over MARC I. MARC I enforced coherency by sharing the actual cache data lines between all Cache Ports, a setup in which the physical ports on the FPGA-internal memory blocks quickly became a bottleneck.

Ports which have been proven to access non-overlapping address ranges using compiler analysis [9] are handled by separate CCs, reducing the amount of intra-cluster coherency traffic. All CCs share a back-end memory bus (MB, i) to the TechMod providing the connection to the physical main memory resources.

MARC II also supports parameterizable support for speculative memory operations in the data path on a per-port basis. Enabling these features uses an approach similar to branch prediction (b) on a conventional processor, but employed here to efficiently allocate the access (g,h,j,k) to shared resources (the intra- and inter-cluster buses, and the single main memory shared with the GPP).

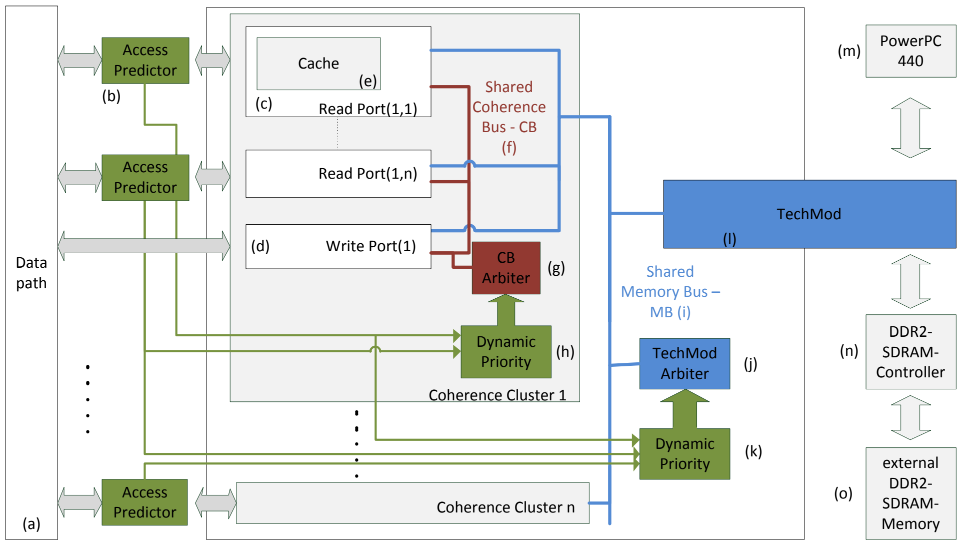


Fig. 1. Overview of the MARC II system

```
int *a, *b, *c, *d;
...
if (*a < 5)
  *d = *a + *b;
else
  *d = *a - *c;
```

Fig. 2. C sample for data path in Fig. 3

III. Support for Speculative Memory Accesses

In a typical SW program, about 20% of the instructions are memory accesses which take up to 100x as much execution time as a simple non-memory instruction [13].

To alleviate this in the spatial computing paradigm used for reconfigurable computing, we propose *memory access speculation* as a novel technique. Memory speculation aims to start executing memory accesses as soon as possible, each on a dedicated memory port. The version presented here specifically relies on *control speculation*, launching accesses as soon as their addresses (for load and store) and data (for stores only) are available, but control conditions have not been evaluated yet. This is similar to branch prediction in GPPs, where instructions are fetched and started from the predicted execution path, to be squashed later in case of a mis-prediction.

We go beyond this approach, which in the memory domain could be seen as a kind of prefetching, and also use speculation statistics to give more likely accesses a higher priority route to shared resources (e.g., CB and MB in MARC II). Our approach collects all required statistics at run-time and does not require compiler support (e.g., execution probability annotations or similar). Since we aim at small hardware areas, we do not exploit yet *data speculation* or speculative writes (which would require complex hardware, e.g., Load Store Queues).

A. Speculation in Temporal and Spatial Compute Models

While explored mostly for GPPs [13], [17], [21], speculative execution is not limited to that domain [3] and can also be used in the parallel paradigms used for HAs [16].

In contrast to the conventional CPU-only architectures, an ACS executes highly parallel calculations on the HA, both in the spatial (e.g., by unrolling) and temporal (pipelining) domains. Here, the parallel data paths rely on equally parallelized memory ports, which feed and drain data to and from the pipelines. The granularity of the memory ports can be as fine as one port per memory operation node in the data path.

While a typical ACS application aims to use as many memory ports as possible to optimally exploit parallelism, such bijective mapping is not mandatory. Multiple operations can also share a single memory port if required due to limited HA resources.

Any hardware architecture aiming to support these highly parallel capabilities has to scale significantly better than the 2 or 3 parallel memory ports of current high-performance GPPs [12]. We achieve this by a distributed architecture, associating speculation features with each individual MARC II Port, instead of relying on a central authority.

Thus, each port is its own speculation domain, allowing both the collection of statistical information as well as the actual speculation-based prioritization on a per-operation basis (assuming a 1:1 mapping of memory operations to Ports).

For Fig. 2, assume software execution on a super-scalar out-of-order GPP: The *a would be loaded, then branch prediction would come in to predict that the resulting value would be less than 5 (assumed here in the example), resulting in starting the load of *b. Once both values arrived, the addition and the store of the result to *d would be launched.

On the RCU with a fully spatial execution model with speculation (Fig. 3), the non-speculative load of *a would be started. In parallel, the speculative loads of *b and *c would also be initiated. After the speculative loads complete, both results (the sum and the product) would be computed, and the control condition (once it becomes available) decides, which value to store to *d.

This approach of speculatively executing all branches of a

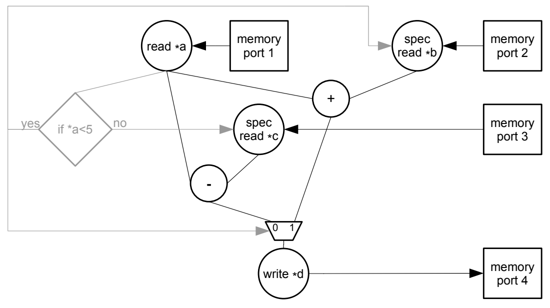


Fig. 3. Parallel data path - speculative execution of memory accesses

conditional in parallel works well, if fully spatial execution on dedicated hardware operators is possible in practice. However, once these dedicated operators actually access a shared resource (a single main memory, in this case), the efficiency deteriorates: On the GPP, the access to *c would never have happened due to branch prediction, resulting in less memory traffic / cache pollution than in the spatial model on the RCU.

B. Lightweight Non-destructive Speculation

Our lightweight approach aims to reach two goals: providing satisfactory speculation speedups while using a minimum of HA resources. By avoiding complex HW structures such as a speculation buffer, which would be required to resolve destructive speculative write operations, significant HW resources can be saved while still providing most of the functionality. Our non-destructive speculation relies on control speculation for all read operations, including intelligent prefetching.

The speculation system, which can be activated for each MARC II Port separately, relies on memory operations to either be *canceled* or *committed* once the control condition has been evaluated. The statistics for each operation's individual behavior is tracked in its associated MARC II port. When speculation support is enabled for a Port, two additional signals are made available to the data path (beyond the usual address and data etc. connections [15]): Output Prepare OP initiates a cached read access from an address, but the data is not finally delivered. If OP is reset at any time, the associated read is canceled (releasing, or possibly not even acquiring the shared resources). On the other hand, if the conventional read signal OE (Output Enable) is set *in addition* to an active OP, the read eventually completes and is thus committed. In all cases, the per-port statistics are appropriately updated. If the Prepare signals are not used, the associated access is handled by MARC II in the usual non-speculative manner.

The individual commit/cancel statistics are fed into one two-level GAg predictor [26], which consists of a **G**lobal pattern history, and a **g**lobal pattern-history table (PHT) containing multiple **A**daptive N-bit predictors. Since the speculative memory system is optionally inserted between the data path and a conventional MARC II Cache Port, it should have only limited impact on the critical path. To this end, the GAg predictor was chosen as the simplest form of a two-level predictor with a single global history and table. Nevertheless, it yields a prediction accuracy of approximately 75% [18] in the SPEC CPU2006 benchmark [22] with 4 bits pattern history length, 2 bits predictor width, and a single memory port for all accesses (CPU).

In the context of the speculative memory system, the value of the predictor represents the probability of actually committing a memory access. After the outcome of the predicted event is known, in this case whether an access was committed or canceled, the N-bit predictor is updated in saturated arithmetic by adding one for a commit and subtracting one for a cancel, and subsequently writing the value back to its place in the PHT.

C. Dynamic Prioritization

The predicted probability of a memory access being actually committed is then used to set the *dynamic priority* of that access for acquiring the shared resources. Instead of the fixed priority scheme used in MARC I, in the speculation-enabled MARC II, all accesses in-flight are sorted by the predicted probabilities by a bitonic sorting network [2]. This combinational parallel circuit determines the order by which accesses travel on the CB and MB.

Note that the priority arrangement of the memory ports may now change dynamically from access to access. Thus, speculative accesses that are unlikely to commit have to wait until speculations with higher commit probability have completed. As the lower-probability accesses have probably been canceled by then, no precious architectural memory bandwidth is wasted on canceled accesses, and high-priority computations are only subject to short delays.

D. Dynamic Prefetching

Dynamic prefetching relies on the predictions, too, but uses them to prefetch lines into the caches of Ports with high commit probabilities *prior* to the actual execution of the memory operation.

Prefetching also depends on the non-speculative/speculative nature of accesses as indicated by the Prepare signals (see Sec. III-B). Whenever a speculative access is initiated, the corresponding cache line is prefetched, subject to the priority determined by the GAg predictors described in the previous Sec.. Hence, a cache line is only prefetched if no other pending accesses (speculative, non-speculative, and prefetch) with higher priority would be hindered. On the other hand, a high-priority prefetch may supersede a low-priority access, which again, by early prefetching, expedites accesses with a high commit probability over those which are unlikely to commit.

If an access leading to a prefetch is canceled by the data path (e.g., once a control condition has been evaluated), and the actual data transfer has already begun, the transfer is cleanly aborted to make the shared buses available to more urgent requests as soon as possible (generally only delaying the new access by an average of just one clock cycle).

IV. Cluster-Coherent Multi-Port Cache

A. Overview

Providing each MARC II Cache Port with its own underlying cache has a number of advantages in our usage scenario. Not only does it allow better scalability (avoiding the bottleneck of actual multi-ported memories), but also interacts better with the speculation mechanism described above (misspeculations on one Port cannot evict correct data held in another Port). The distributed scheme also fits better with the spatial computation paradigm and allows a higher degree of parallelism than the MARC I approach of physically sharing the cache lines between the Ports.

B. Cache Architecture

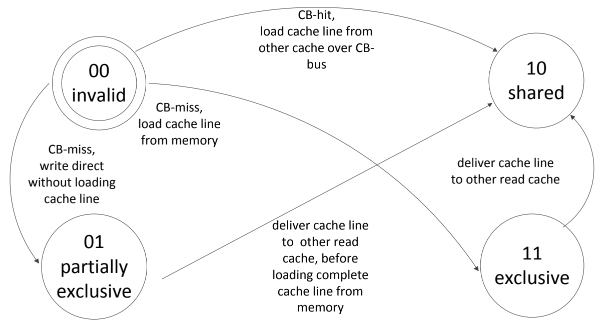
The caches themselves are implemented using BlockRAMs, with the parametrized line length being a multiple of 256b (eight 32b words). This is due to the 128b wide external memory interface (see Sec. V), running at double-data rate relative to MARC II. In the minimal configuration, eight 32b wide BlockRAMs are used per Cache Port. Based on our experience with MARC I, the MARC II caches are direct mapped instead of fully associative: While direct mapping has a lower efficiency than higher-degree caches [8], the abundance of BlockRAMs on modern FPGAs allows better scaling than implementing the larger CAMs required for higher associativity.

C. Coherency Mechanisms

Coherency among the distributed caches is managed in two ways: If static points-to-analysis can prove that two accesses will not affect the same address ranges, these independent accesses will be assigned to Cache Ports in different CCs. No explicit coherency management between them is required. Potentially dependent accesses are assigned to Cache Ports in the same Cluster, with explicit coherency mechanisms (in contrast to, e.g., [19], which provided only incoherent caches).

We make design decisions to reduce the coherency-maintaining intra-cluster traffic (and simplify the underlying hardware): First, since we aim to use one Port per access, we have separate Ports for loading and storing. Second, we allow only one Write Port per CC. This reflects that even in sequential code, e.g., for SPECint 2000, more than double the number of loads than stores is performed. If more dependent stores occur, these are processed either sequentially on the single Write Port, or by realizing one CC using a MARC I-like shared cache (which realistically scales-up to four parallel accesses of any kind, e.g., three Write and one Read Ports).

With these choices, we can now replace the potentially slow MSI, MESI, or MOESI coherence protocols [1], [11] with the simpler approach shown in Fig. 4. Cache lines in a Read Port are either *valid* or *invalid*, requiring just 1b of state. Write Ports are either *invalid* (the cache line is not present), *shared* (the cache line is present, and also present in at least one other Write Cache), *exclusive* (the cache line is present and no other cache has it), and *partially exclusive* (the cache line is only partially present). Note that the last state is only needed for



CB-hit, load cache line from other cache over CB-bus
00 invalid
10 shared
CB-miss, load cache line from memory
CB-miss, write direct without loading cache line
deliver cache line to other read cache
01 partially exclusive
deliver cache line to other read cache, before loading complete cache line from memory
11 exclusive

Fig. 4. Cache line state transitions in a Write Port

the write-without-allocate policy (Sec. IV-D). In Fig. 4, we do not show the back-edges from all states to *invalid*, which are taken whenever a cache is flushed or invalidated. Note that the usual *modified* state is not required here, since a Write Port's cache only holds modified lines at all.

The intra-CC Coherency Bus (CB) is used to update states and cache lines between Ports. It consists of a cache line address (e.g., 10b for a configuration of 1024 lines of 8 32b words each) and a tag (17b, in the same case), 256b of data, as well as data and address valid signals. Access to the CB is managed by the CB Arbiter using request/grant signals (optionally guided by Dynamic Prioritization, Sec. III-C).

The way the states are used can be set for each CC separately: For a CC in invalidate mode, the Write Port notifies the Read Ports holding an affected shared cache line to invalidate it. If the Read Ports later require the cache line again, it will be retrieved over the CB from the Write Port. In update mode, the Write Port immediately sends out its altered shared cache line over the CB to all Read Ports holding old versions. The best configuration can be chosen at compile time, using either static analysis or by evaluating dynamic memory traces.

D. Write-without-Allocate

The Memory Bus (MB), having a 32b address and 256b data component, is shared between all CCs. To avoid it becoming a bottleneck, the cache in the Write Port can accept a write, but does not fetch (allocate) the corresponding cache line. Instead, it keeps track of the specific words written in the cache line. Only the words actually modified will be written back to main memory (selected using the 32 byte enable signals on the MB). When a Read Port requests a cache line partially modified by the Write Port, the original contents from main memory are merged on-the-fly with the Write Port's modifications without incurring additional latency. For the common use-case of the HA writing results to memory, which are read later only by software on the GPP (but not using Read Ports in the HA), this approach reduces pressure on the MB.

E. Priority-based Bus Arbitration

When enabling control speculation for a Port, both the CB and MB bus arbiters are controlled using the dynamic priority

Name	%Writes	Ran-dom?	1-CC Update	1-CC Inval.	2-CC (Update)
L1	0	N	3.758	3.758	5.695
L2	50	N	4.495	4.732	6.053
L3	100	N	5.104	5.705	6.605
L4	0	Y	5.642	5.642	11.193
L5	50	Y	6.196	6.200	10.338
L6	100	Y	6.569	6.509	10.914

TABLE I
LINKED LIST: MILLION EXECUTION CLOCK CYCLES

Name	Addr a	Addr b	Addr c	partial write?	inval/ update	# cycles
A01	0x0000	0x1000	0x2000	Y	–	3507
A02	0x0000	0x1000	0x2000	N	–	4309
A03	0x1000	0x1000	0x0000	Y	–	2446
A04	0x1000	0x1000	0x0000	N	–	3469
A05	0x0000	0x0000	0x0000	–	I	4072
A06	0x0000	0x0000	0x0000	–	U	2866
A07	0x0000	0x1000	0x0000	–	I	3612
A08	0x0000	0x1000	0x0000	–	U	2887
A09	0x0000	0x0000	0x0000	–	I	1251
A10	0x0000	0x0000	0x0000	–	U	1291

TABLE II
ARRAY PROCESSING: OVERLAPPING/NON-OVERLAPPING

appropriate for the specific access (see Sec. III-C). Very likely accesses will thus acquire the shared resources quicker than unlikely accesses.

F. Canceling Misspeculated Accesses

Misspeculated accesses, indicated at the Port interface using the Prepare signals (see Sec. III-B), are recalled as quickly as possible to keep them from loading the shared resources: If an access has requested, but not yet acquired CB, access, the request is withdrawn. MB access is handled similarly. The capability also extends to the TechMod interface to main memory: Instead of waiting for the reply to a misspeculated read to arrive, the reply is marked to be discarded in the TechMod and the next potentially useful access is forwarded immediately to the main memory controller.

V. IMPLEMENTATION

An initial implementation of MARC II was done on the Xilinx ML507 platform [25]. The Virtex 5 FX FPGA realizes a single-chip ACS architecture, combining the embedded PowerPC 440 GPP with HA(s) on the reconfigurable fabric. We modified the Reference Design by inserting the TechMod into the MCI bus between PowerPC and external 200 MHz DDR2-SDRAM main memory. The GPP can thus access the HA in a simple memory mapped fashion, while the 100 MHz HA can directly access main memory, following our FastLane architecture [16] of giving the GPP priority over the HA to ensure stable system operation. The system was implemented using Xilinx EDK and ISE 10.3, and Synplify Premier DP 9.6.1.

VI. EXPERIMENTAL EVALUATION

To evaluate the effects of the different mechanisms, we used a number of micro-benchmarks to examine the interaction of specific features. Once we have targeted our high-level-language to ACS compiler to MARC II (see Sec. VIII), we will be able to run full application-level benchmarks.

A. Cache Operations

Table I shows the result of processing a linked list of 128K four-word elements using one Write Ports (WP) and three Read Ports (RP). The list is laid out either sequentially and randomly in memory. The processing is varied from just reading to also writing in every iteration, with a mix in between. Furthermore, we run the operation both with just one CC, and then with two CCs processing different lists in parallel. The coherency strategy varies between Update and Invalidate modes. While increasing randomness (=cache misses) and writes (=more coherency traffic) grow the runtime, the performance degrades only slowly. Also, note that when only reading, the coherency strategy is irrelevant. For mostly sequential accesses with writes (L2,3), Update is better (=allows many cache hits). This advantage shrinks with increasing randomness (L5), both Invalidate and Update have nearly the same performance then. With even more writes (L6) and randomized layout, the efforts of Update are wasted (since the updated cache lines will not be read again soon), and Invalidate performs better. Furthermore, running the application on two CCs in parallel leads to a net speed-up (despite sharing the MB, the execution times do not double).

A second benchmark examines operations in overlapping (=requiring coherency handling within a CC) and non-overlapping (=executable in independent CCs) array ranges. Each of the arrays is assumed to contain 256 32b elements (thus running entirely out of cache, to disregard main memory latency) and is processed using the operation $c[i] = a[i] + b[i-1]$, requiring one WP and two RPs. The benchmark examines different overlaps between the read and written arrays, as well as combinations of the partial write feature and the Invalidate/Update coherency strategies. In A01 to A04 we show the benefits of the partial write feature. It can save up to 41% (A03 vs. A04) if the written data is not read again. A03,A04 are faster than A01,A02 since the overlapping read arrays are handled using cache-to-cache transfers. Note that we have yet to find an application where partial writes are not beneficial, we thus suggest enabling it as default. A05 to A10 compare the Invalidate/Update coherence strategies: If the written and read arrays overlap, Update allows the sequential reads to also hit the caches more often. A09 and A10 are altered to have index stride of eight to examine reducing this locality, provoking more misses. For such cases, the Invalidate strategy is a slightly better choice. For reference, this entire HA requires 8536 LUTs and 26 BlockRAMs on the Virtex 5 FPGA.

B. Speculation

To test the effect of speculation, we consider four micro-benchmarks, see Table III, all in a single CC. First, we examine the impact of speculation on the *L5 Linked List* benchmark.

Name	w/ spec.	w/o spec.	speed-up
Linked List (3RP 1WP)	6.196	6.402	1.03x
Tree Search (2RP)	10.011	11.087	1.11x
Merge Sort (2RP 1WP)	10.266	59.498	5.80x
COMRADE Sample (2RP)	0.356	0.598	1.67x

TABLE III
SPECULATION: MILLION EXECUTION CLOCK CYCLES

Due to the random nature, only limited gains are achievable (mostly due to dynamic prefetching) with speculation enabled. *Tree Search* searches for random keys in a 16,384-element binary tree. It uses two RPs, simultaneously speculatively reading both the left and right successors of a node. The actual control condition is assumed to require 16 clock cycles for evaluation (e.g., a string comparison). The speculation allows the hiding of the main memory latencies, but is limited to just 10% improvement when searching for random keys. Note that this will significantly improve when ordered sequences of keys are being searched for. In that case, the PHT predictors precompute the correct path through the tree and lead to the prioritization of the correct memory accesses. *Merge Sort* is the inner loop of the CoreMark MergeSort benchmark [5], which sorts a list of 139807 elements by merging the sorted sub-lists in-place. This control-intensive code profits much more from speculation, specifically from dynamic prefetching, which again hides the 16 cycles latency of the element-wise comparison. Finally, *COMRADE Sample* sums values from two arrays: Every third number is taken from one array, the two others from the second array. This benchmark has actually been automatically compiled from C by the COMRADE ACS compiler [6] into a HA, which was then manually fitted with the MARC II system. COMRADE already supports powerful speculation mechanisms, but currently targets MARC I, thus the need for manual intervention.

VII. RELATED WORK

Shared memory parallel processing on FPGAs is still relatively uncommon. [19] used distributed caches, but did not address coherency at all. This was treated in [23], but with more complex hardware (due to combined read/write ports), and lack of separate coherence clusters and cache-to-cache transfers. More work has been performed on keeping multiple soft-processor cores coherent [10], but this used relatively slow software synchronization. To our knowledge, no prior attempts at using prediction techniques to optimize memory accesses in spatial computing exist. For superscalar GPPs, the distantly related use of branch prediction data to optimize loads on misspeculated paths has been proposed in [20] [4].

VIII. CONCLUSION AND FUTURE WORK

MARC II considerably enhances the capability of both our own as well as other memory systems for reconfigurable computers. It is the first one to combine support for control speculation, distributed memories, and clustered coherence mechanisms, with the FastLane system interface to GPP and main memory.

Our next research will proceed in two directions: First, we will enhance our high-level language to ACS compilers to target MARC II and automatically configure it as appropriate for the current application (e.g., assignment of Ports to CCs, select coherence strategy, etc.). Second, we will extend MARC II itself, both using local improvements (e.g., low-degree associativity, victim line handling, etc.) as well as providing full support for speculative writes.

REFERENCES

[1] AMD Inc., "AMD64 Architecture Programmer's Manual Vol 2: System Programming", 2007.

[2] Batcher K. E., "Sorting networks and their applications", *AFIPS Conf. Proc. 32*, 1968.

[3] Bobba J., Goyal N., Hill M. D., Swift M. M., Wood D. A., "TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory", *Proc. 35th Intl. Symp. on Comp. Arch.*, 2008

[4] Ebrahimi E., et al., "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems", *Proc. Intl. Symp. on High-Perf. Comp. Arch.*, 2009

[5] EEMBC, "What is CoreMark?", *www.coremark.org*, 2009

[6] Gädke, H., Stock, F., Koch, A., "Memory Access Parallelization in High-Level Language Compilation for Reconfigurable Adaptive Computers", *Proc. Intl. Conf. on Field Programmable Logic*, 2008

[7] Gädke-Lütjens, H., Thielmann, B., Koch, A., "A Flexible Compute and Memory Infrastructure for High-Language to Hardware Compilation", *Proc. Intl. Conf. on Field-Programmable Logic*, 2010.

[8] Hennessy, J.L., Patterson, D.A., "Computer Architecture: A Quantitative Approach", *Morgan Kaufmann*, 2006

[9] Hind, M., Pioli, A., "Which pointer analysis should I use?", *Proc. Intl. Symp. on SW Testing and Analysis*, 2000

[10] Hung, A. et al., "Symmetric Multiprocessing on Programmable Chips Made Easy", *Proc. DATE*, 2005

[11] Intel Corp, "Intel 64 and IA-32 Architectures Optimization Reference Manual", March 2009.

[12] Intel Corp., "First the Tick, Now the Tock: Intel Microarchitecture (Nehalem)", *White Paper No. 319724*, 2009.

[13] Kaeli D. R., Yew P.-C., "Speculative execution in high-performance computer architectures", *Chapman & Hall/CRC*, 2005.

[14] Koch, A., "Advances in Adaptive Computer Technology", *habilitation thesis*, TU Braunschweig (Germany), 2004

[15] Lange, H., Koch, A., "Memory Access Schemes for Configurable Processors", *Proc. Intl. Conf. on Field-Programmable Logic*, 2000

[16] Lange H., Koch A., "Architectures and Execution Models for Hardware/-Software Compilation and their System-Level Realization", *IEEE Trans. on Computers*, IEEE Digital Lib., 12-2009.

[17] Marcuello P., Gonzalez A., Tubella J., "Speculative multithreaded processors", *Proc. 12th Intl. Conf. on Supercomputing*,1998

[18] Martynus M., "Branch Predictor Simulator", *Bachelor Thesis, TU Darmstadt (Germany)*, 2007.

[19] Putnam, A., et al., "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures", *Proc. Intl. Conf. on Field-Programmable Logic*, 2008

[20] Sendag, R., et al., "Exploiting the Prefetching Effect Provided by Executing Mispredicted Load Instructions", *Proc. Euro-Par Conf.*, 2002

[21] Shen J. P., Lipasti M., "Modern Processor Design", *McGraw-Hill*, 2005.

[22] SPEC CPU Subcommittee, original program authors, "SPEC CPU2006 Benchmark Descriptions", *SPEC - Standard Performance Evaluation Corp.*, 2006

[23] Woods, D. "Coherent Shared Memories for FPGAs", *Master Thesis, U Toronto*, 2009

[24] XILINX Inc, "LogiCORE IP DDR2 Memory Controller for PowerPC 440", 2010

[25] XILINX Inc, "ML505/ML506/ML507 Reference Design User Guide", *UG 349*, 2009

[26] Yeh T. Y., Patt Y. N., "Alternative Implementations of Two-Level Adaptive Branch Prediction", *Proc. 19th Intl. Symp. on Comp. Arch. (ISCA)*, 1992.